

# Probabilistic Programming Languages

Guillaume Baudart

*MPRI 2024-2025*



# Warm-up: Hidden Markov Model

---

Probabilistic Programming Languages

# HMM: Hidden Markov Model

Track the position of an agent from noisy observations

- The current position should not be too far from the previous position
- The observations should not be too far from the current position

Probabilistic model:  $\forall t \in \mathbb{N}$

- $x_t \sim \mathcal{N}(x_{t-1}, \text{speed})$
- $y_t \sim \mathcal{N}(x_t, \text{noise})$

Try it in BYO-PPL!

# HMM: Hidden Markov Model

*hmm.ml*

```
open Basic.Importance_sampling

let hmm prob data =
  let rec gen states data =
    match (states, data) with
    | [], y :: data → gen [ y ] data
    | states, [] → states
    | pre_x :: _, y :: data →
      let x = sample prob (gaussian ~mu:pre_x ~sigma:1.0) in
      let () = observe prob (gaussian ~mu:x ~sigma:1.0) y in
      gen (x :: states) data
  in
  gen [] data

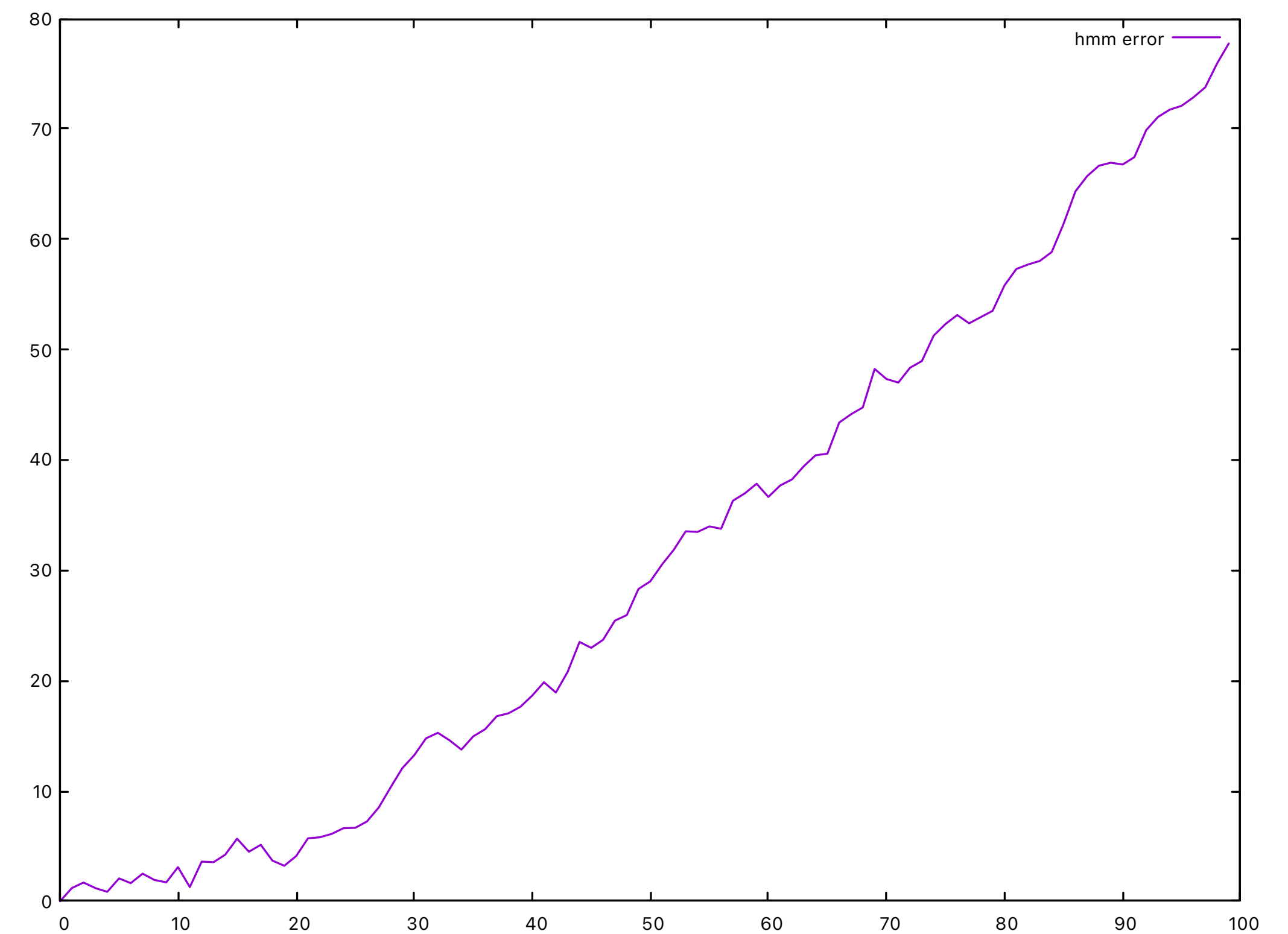
let _ =
  let data = List.init 20 (fun i → Float.of_int i) in
  let dist = Distribution.split_list (infer hmm data) in
  let m_x = List.rev (List.map Distribution.mean dist) in
  List.iter2 (Format.printf "%f >> %f@.") data m_x
```

# HMM: Hidden Markov Model

```
› dune exec ./examples/hmm.exe
```

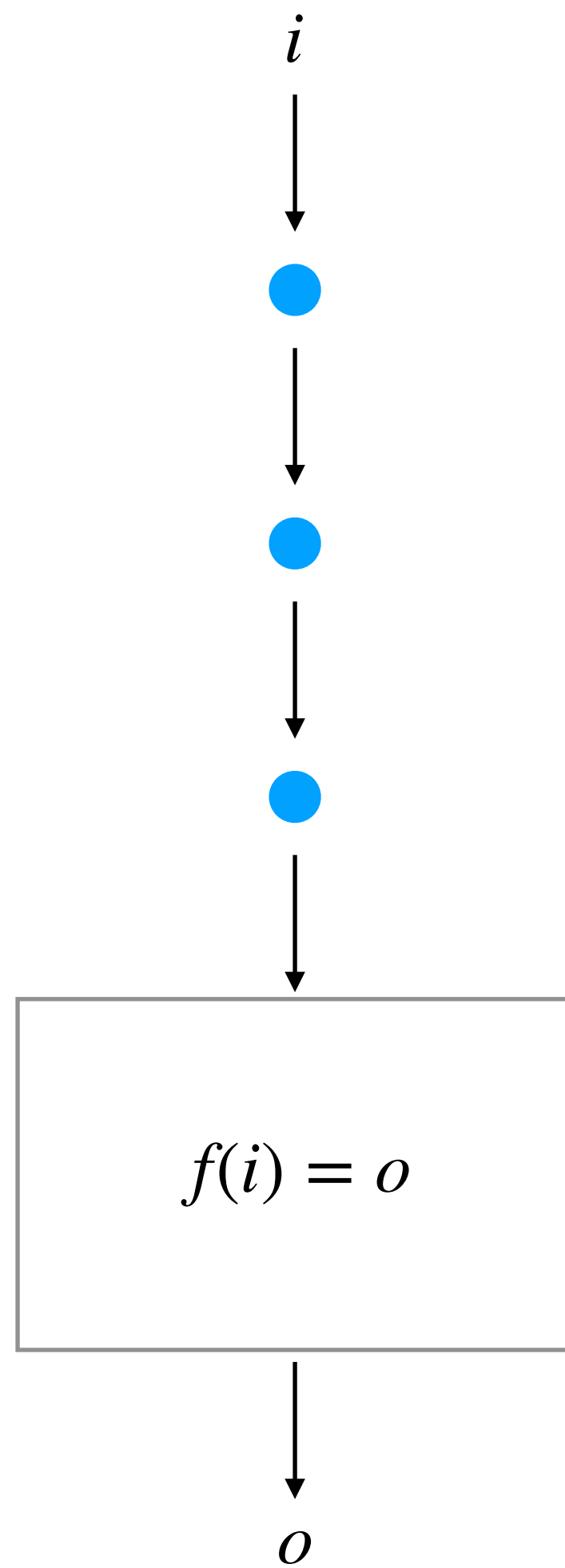
```
0.000000 >> 0.000000
1.052632 >> 0.278989
2.105263 >> 2.923428
3.157895 >> 2.812035
4.210526 >> 2.328341
5.263158 >> 1.742109
6.315789 >> 2.518105
7.368421 >> 3.958375
8.421053 >> 5.946233
9.473684 >> 7.329554
10.526316 >> 9.293653
11.578947 >> 10.181831
12.631579 >> 8.549409
13.684211 >> 9.323073
14.736842 >> 9.280692
15.789474 >> 9.352218
```

...

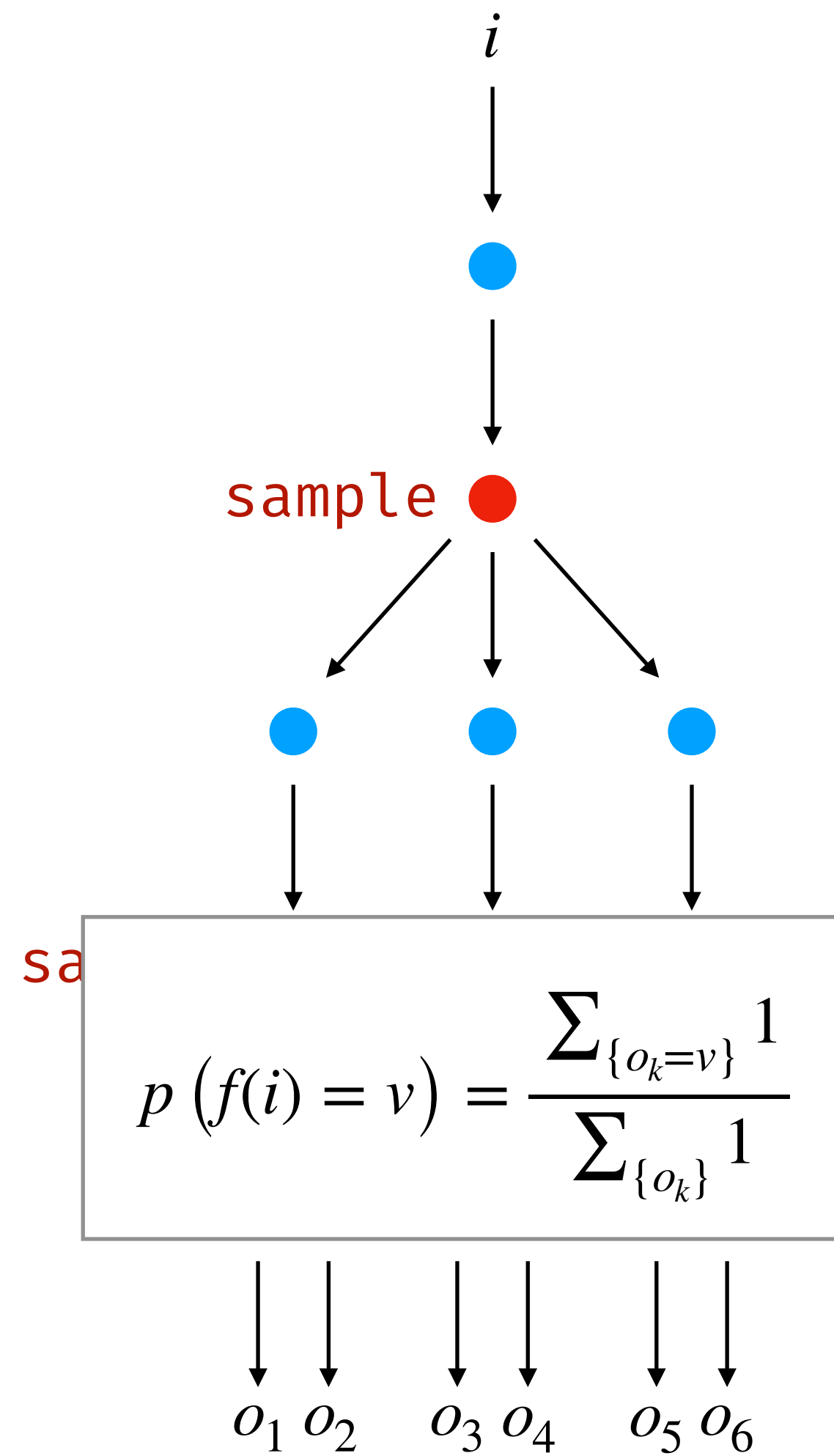


**infer** :  $(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$  dist

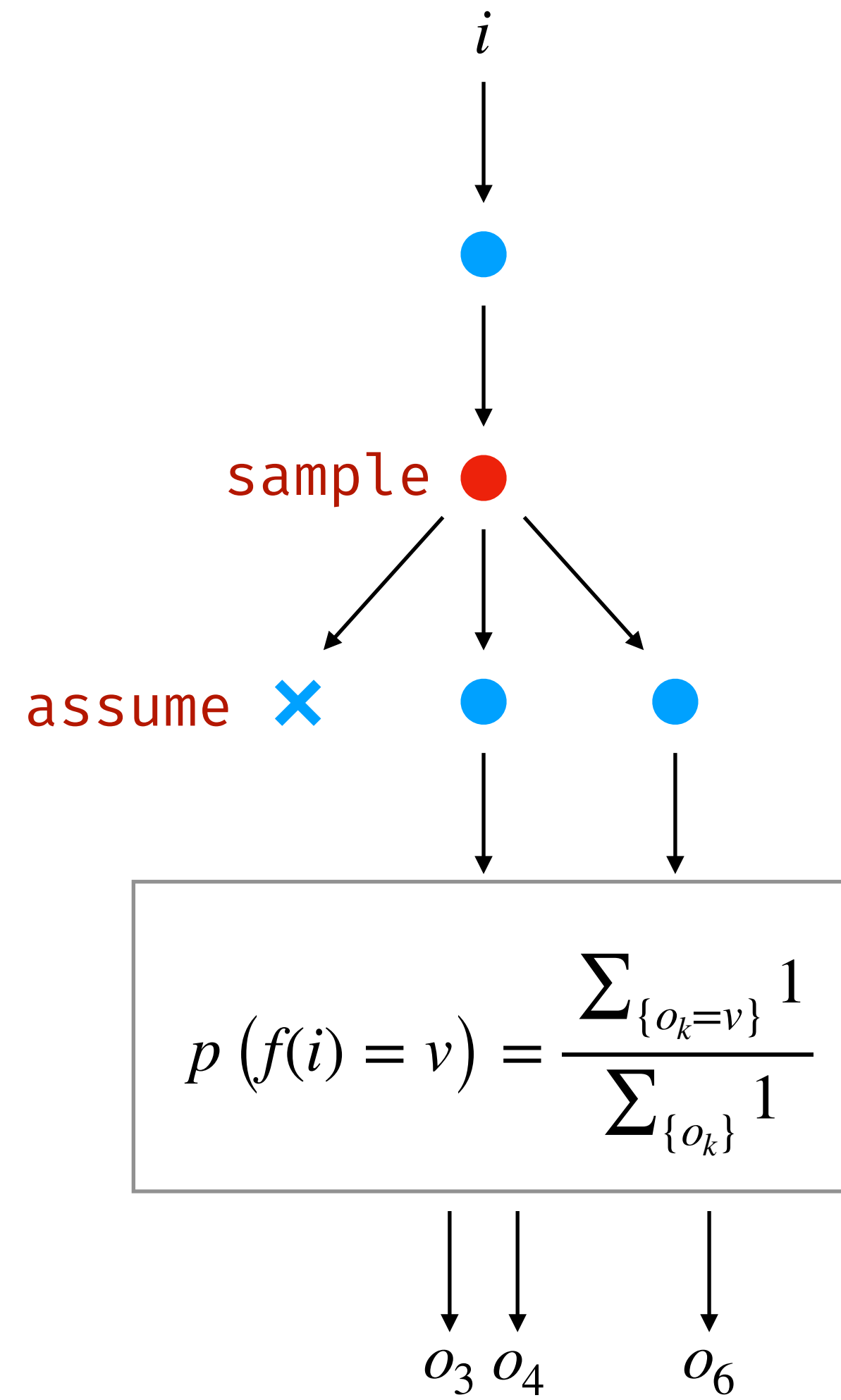
program



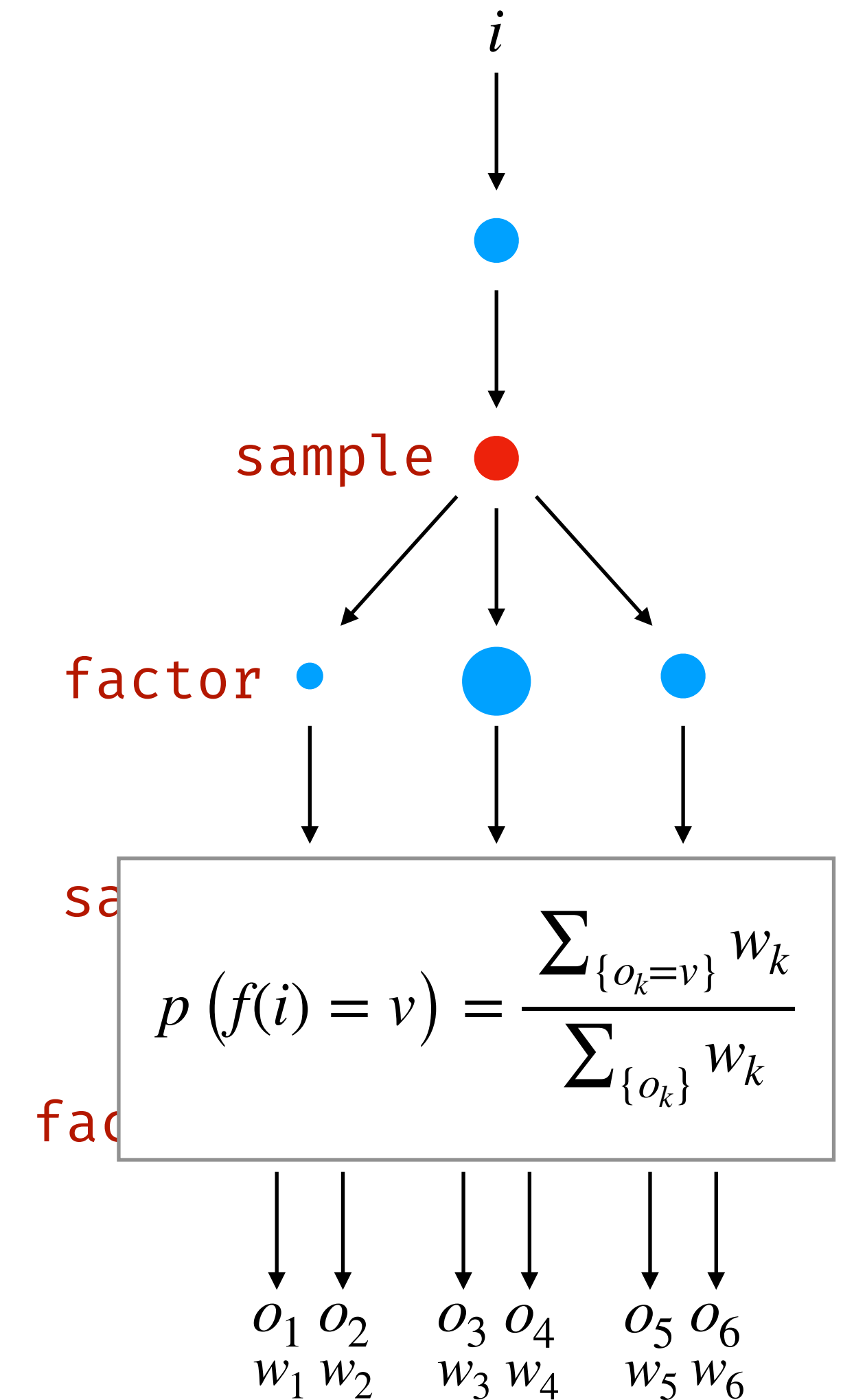
sample



assume



factor



# HMM: Importance sampling

## Problem:

- Each particle does a random walk and compute a score
- No re-calibration based on observations
- Score decreases at each factor statement



# HMM: Importance sampling

## Problem:

- Each particle does a random walk and compute a score
- No re-calibration based on observations
- Score decreases at each factor statement

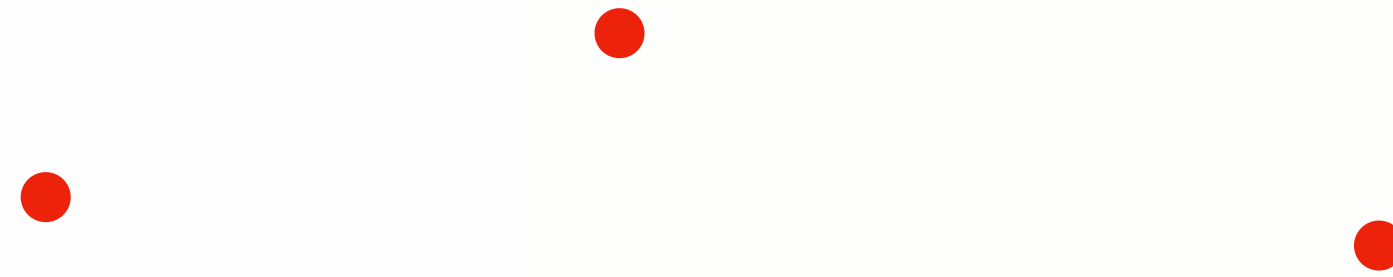




# HMM: Importance sampling

## Problem:

- Each particle does a random walk and compute a score
- No re-calibration based on observations
- Score decreases at each factor statement



# HMM: Importance sampling

## Problem:

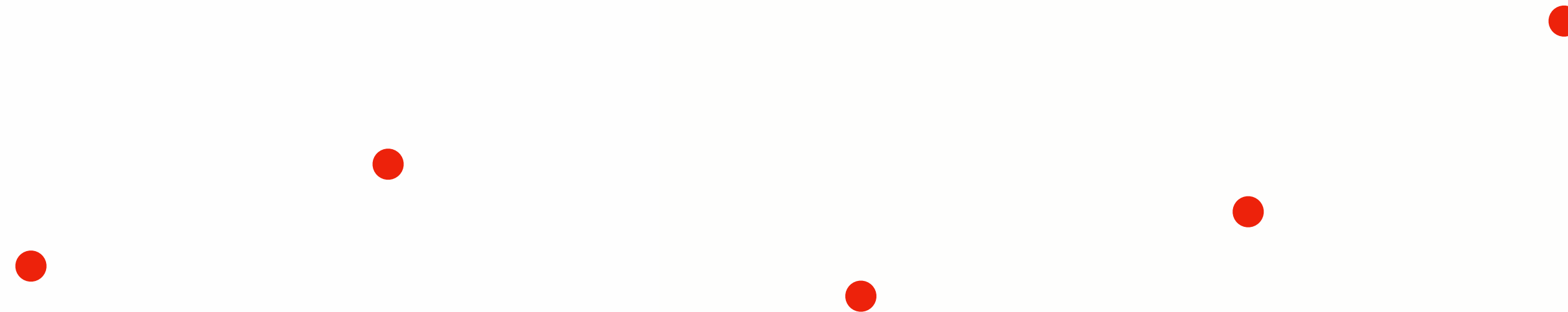
- Each particle does a random walk and compute a score
- No re-calibration based on observations
- Score decreases at each factor statement



# HMM: Importance sampling

## Problem:

- Each particle does a random walk and compute a score
- No re-calibration based on observations
- Score decreases at each factor statement



Bad estimation

# Remember: The curse of dimensionality

Problem becomes harder as the dimension increases

Basic inference: rejection sampling, importance sampling

- Performances decrease exponentially when the dimension increases
- Only use for low-dimension models

17h45mn

How to mitigate this problem?

- Make assumptions about the posterior distributions
- Break the problem into simpler, smaller problems



# HMM: Particle filter

Add a resampling step

- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution





# HMM: Particle filter

Add a resampling step

- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution



# HMM: Particle filter

Add a resampling step

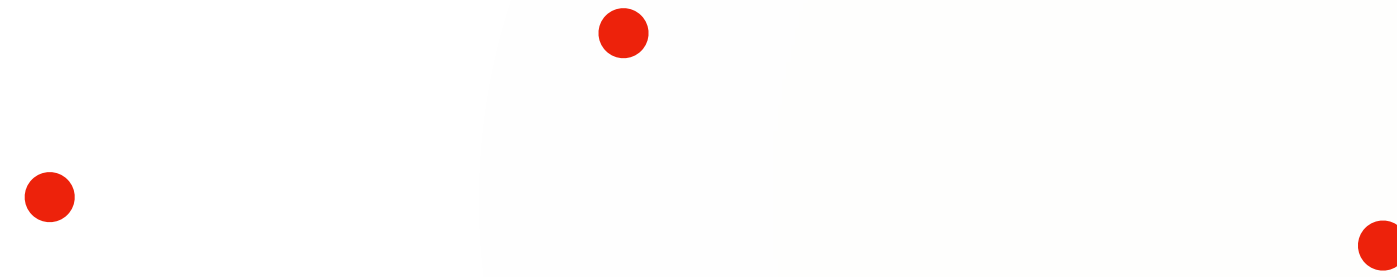
- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution



# HMM: Particle filter

Add a resampling step

- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution

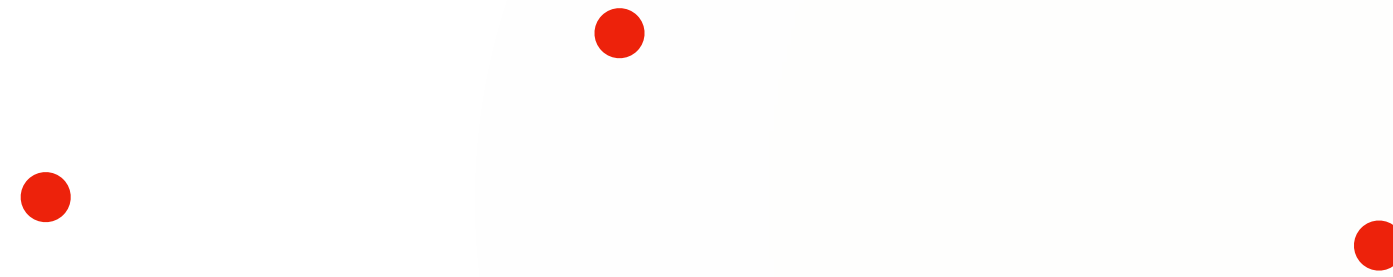




# HMM: Particle filter

Add a resampling step

- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution



# HMM: Particle filter

Add a resampling step

- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution



# HMM: Particle filter

Add a resampling step

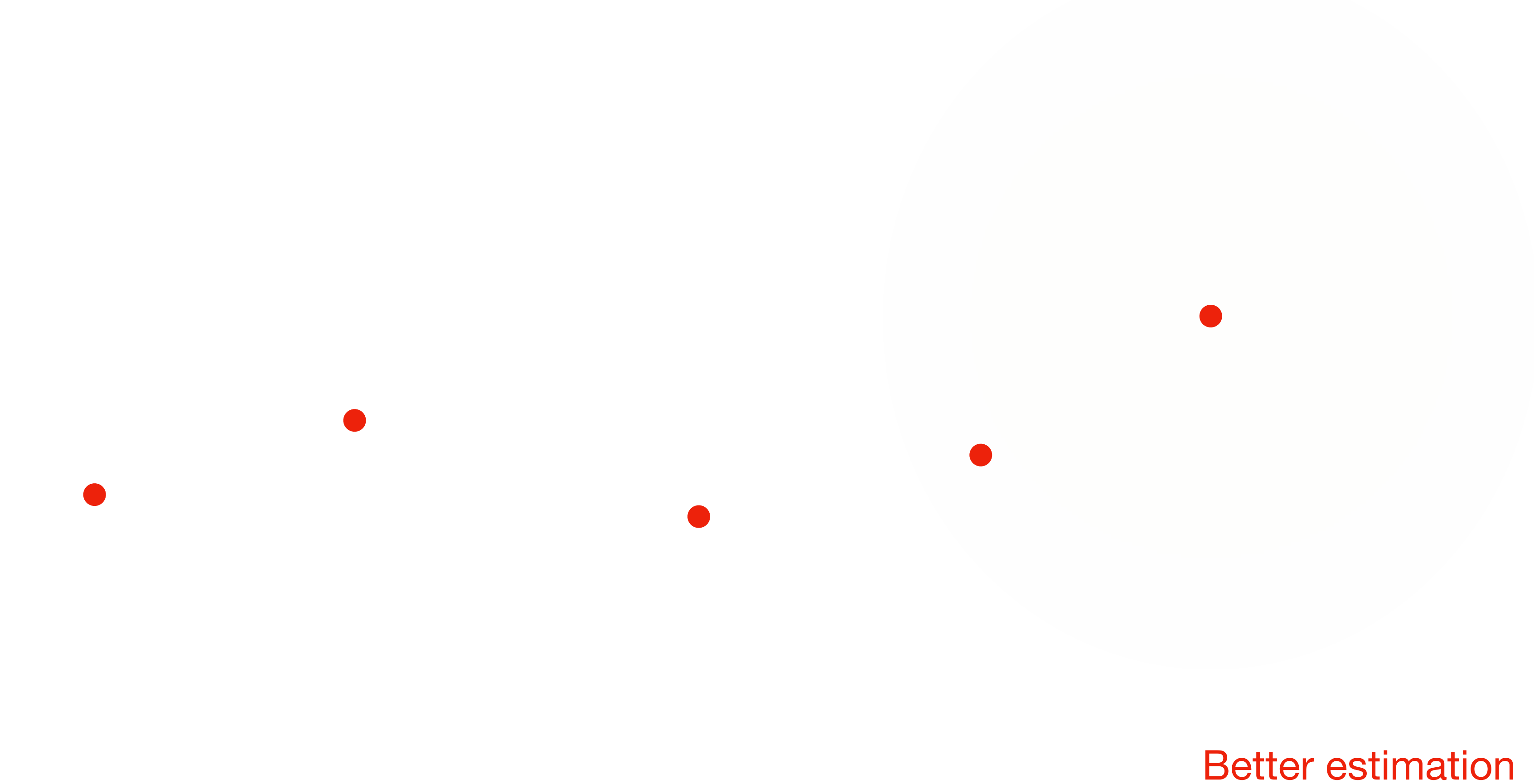
- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution



# HMM: Particle filter

Add a resampling step

- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution



# Problem: Duplications

How can we duplicate a particle during execution?

- Rerun the particle from the start?
- Force reuse sampled values?
- Clone the memory state?

## Continuation Passing Style

- Functions take an extra argument `k`: the continuation
- `k` implements what should be done with the result of the function
- In our context, we can use continuation to interrupt/restart the execution of a model

# Continuation Passing Style (CPS)

---

Probabilistic Programming Languages

# Reminders: CPS

*tree.ml*

```
let rec tree_height t =  
  match t with  
  | Empty → 0  
  | Node (_, l, r) → 1 + max (tree_height l) (tree_height r)
```

# Reminders: CPS

*tree.ml*

```
let rec tree_height t =  
  match t with  
  | Empty → 0  
  | Node (_, l, r) → 1 + max (tree_height l) (tree_height r)
```

```
let rec tree_height t =  
  match t with  
  | Empty → 0  
  | Node (_, l, r) →  
    let hl = tree_height l in  
    let hr = tree_height r in  
    (1 + max hl hr)
```

1. Add intermediate values



# Reminders: CPS

*tree.ml*

```
let rec tree_height t =  
  match t with  
  | Empty → 0  
  | Node (_, l, r) → 1 + max (tree_height l) (tree_height r)
```

```
let rec tree_height t k =  
  match t with  
  | Empty → k 0  
  | Node (_, l, r) →  
    let hl = tree_height l in  
    let hr = tree_height r in  
    k (1 + max hl hr)
```

1. Add intermediate values
2. Add call to continuation

# Reminders: CPS

*tree.ml*

```
let rec tree_height t =  
  match t with  
  | Empty → 0  
  | Node (_, l, r) → 1 + max (tree_height l) (tree_height r)
```

```
let rec tree_height t k =  
  match t with  
  | Empty → k 0  
  | Node (_, l, r) →  
    tree_height l (fun hl →  
      tree_height r (fun hr →  
        k (1 + max hl hr)))
```

1. Add intermediate values
2. Add call to continuation
3. Turn let/in into nested function call

# Funny bernoulli CPS

*funny\_bernoulli.ml*

```
let funny_bernoulli () =  
  let a = sample (bernoulli ~p:0.5) in  
  let b = sample (bernoulli ~p:0.5) in  
  let c = sample (bernoulli ~p:0.5) in  
  let () = assume (a = 1 || b = 1) in  
  a + b + c
```

1. Add intermediate values
2. Add call to continuation
3. Turn let/in into nested function call

# Funny bernoulli CPS

*funny\_bernoulli.ml*

```
let funny_bernoulli () =  
  let a = sample (bernoulli ~p:0.5) in  
  let b = sample (bernoulli ~p:0.5) in  
  let c = sample (bernoulli ~p:0.5) in  
  let () = assume (a = 1 || b = 1) in  
  a + b + c
```

```
let funny_bernoulli () k =  
  sample (bernoulli ~p:0.5) (fun a →  
    sample (bernoulli ~p:0.5) (fun b →  
      sample (bernoulli ~p:0.5) (fun c →  
        assume (a = 1 || b = 1) (fun () →  
          k (a + b + c))))
```

1. Add intermediate values
2. Add call to continuation
3. Turn let/in into nested function call

# CPS monadic operators

*cps\_operators.ml*

```
let return e k = k e
```

```
let ( let* ) e f k = e (fun x → f x k)    (* let* x = e in f(x) *)
```

# CPS monadic operators

*cps\_operators.ml*

```
let return e k = k e
```

```
let ( let* ) e f k = e (fun x → f x k)    (* let* x = e in f(x) *)
```

```
let funny_bernoulli () k =  
  sample (bernoulli ~p:0.5) (fun a →  
    sample (bernoulli ~p:0.5) (fun b →  
      sample (bernoulli ~p:0.5) (fun c →  
        assume (a = 1 || b = 1) (fun () →  
          k (a + b + c))))
```

```
let funny_bernoulli () =  
  let* a = sample (bernoulli ~p:0.5) in  
  let* b = sample (bernoulli ~p:0.5) in  
  let* c = sample (bernoulli ~p:0.5) in  
  let* () = assume (a = 1 || b = 1) in  
  return (a + b + c)
```

Sample generation (CPS)

---

Probabilistic Programming Languages

# CPS models

*infer.ml*

```
module Gen : sig
  type 'a prob
  and 'a next = 'a prob → 'a prob
  and ('a, 'b) model = 'a → ('b → 'b next) → 'b next

  val sample : 'a Distribution.t → ('a → 'b next) → 'b next
  val factor : float → (unit → 'b next) → 'b next
  val draw: ('a, 'b) model → 'a → 'b
end = struct ... end
```

## Type 'a prob

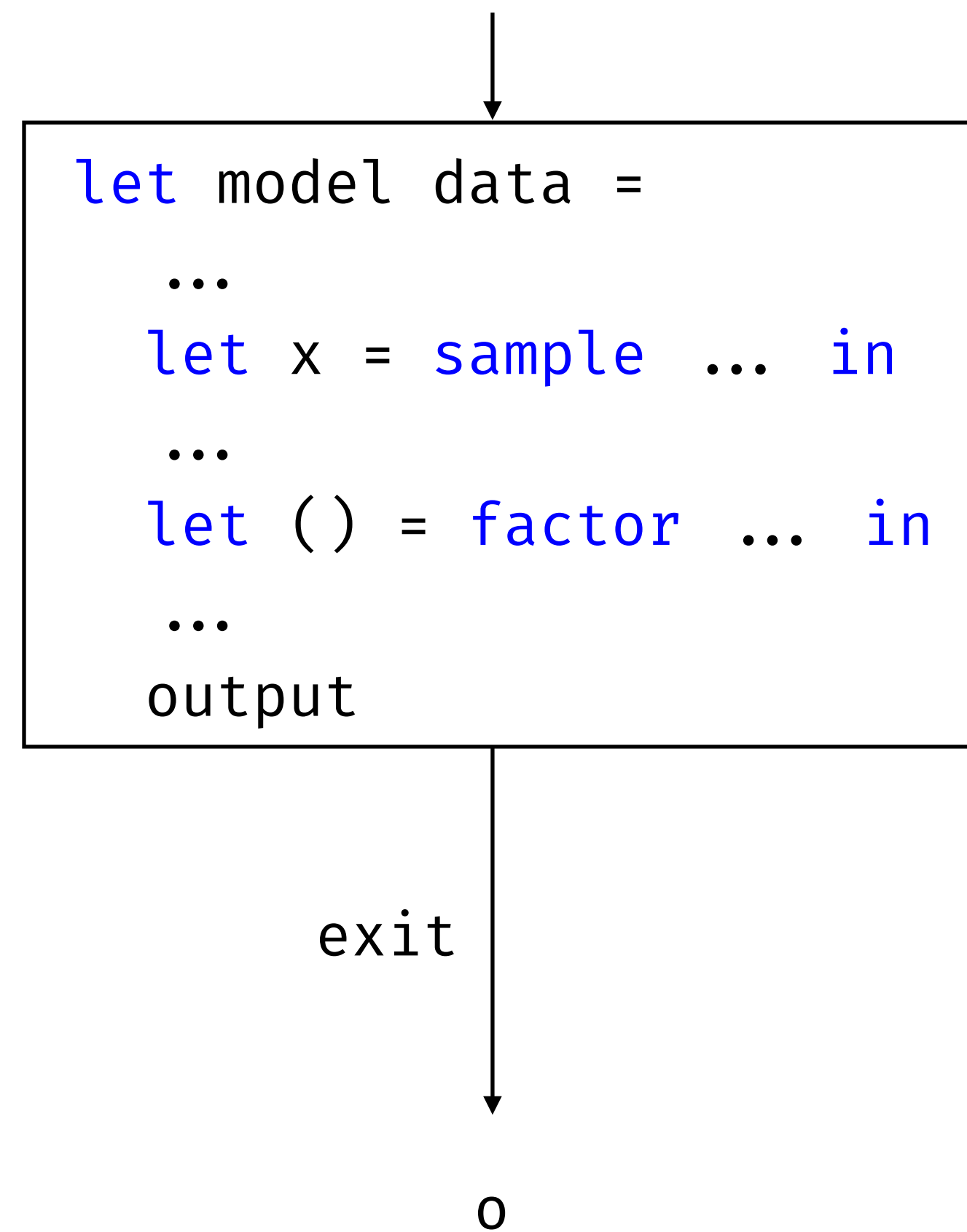
- Store all information required for inference
- Type ('a, 'b) model capture input/output types

## Models where probabilistic constructs are CPS functions

- Two arguments: input 'a and a continuation on the return value ('b → 'b next).
- The return value is a continuation 'a next that updates a probabilistic state of type 'a prob.



# Sample generation



# Sample generation

*infer.ml*

```
module Gen = struct
  type 'a prob = 'a option
  and 'a next = 'a prob → 'a prob
  and ('a, 'b) model = 'a → ('b → 'b next) → 'b next

  let exit v _prob = Some v                                     (* return value *)

  let sample d k prob =
    let v = Distribution.draw d in                               (* draw *)
    k v prob                                                     (* continue *)

  let factor _s k prob = k () prob                             (* continue, ignore score *)

  let draw model data =
    let v = (model data) exit None in                           (* run model *)
    Option.get v                                                 (* extract value *)

end
```

# Funny bernoulli

*funny\_bernoulli.ml*

```
open Infer.Gen
```

```
let funny_bernoulli () =  
  let* a = sample (bernoulli ~p:0.5) in  
  let* b = sample (bernoulli ~p:0.5) in  
  let* c = sample (bernoulli ~p:0.5) in  
  let* () = assume (a || b) in  
  return (Bool.to_int a + Bool.to_int b + Bool.to_int c)
```

```
let _ =  
  for _ = 1 to 10 do  
    let v = draw funny_bernoulli () in  
    Format.printf "%d " v  
  done
```

---

```
> dune exec ./examples/funny_bernoulli.exe
```

```
1 1 2 2 2 2 2 1 3 2
```

Importance sampling (CPS)

---

Probabilistic Programming Languages

# Importance sampling

*infer.ml*

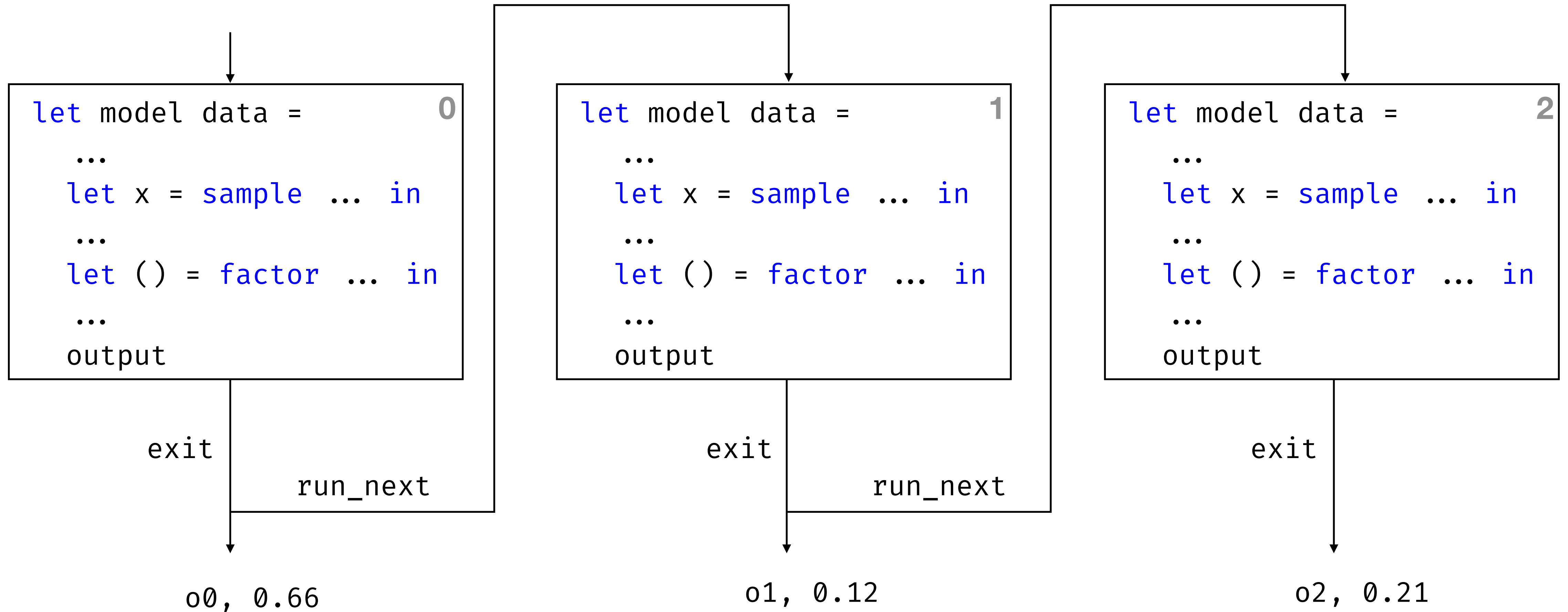
```
module Importance_sampling : sig
  type 'a prob = { score : float; k : 'a next; value : 'a option }
  and 'a next = 'a prob → 'a prob
  and ('a, 'b) model = 'a → ('b → 'b next) → 'b next

  val sample : 'a Distribution.t → ('a → 'b next) → 'b next
  val factor : float → (unit → 'b next) → 'b next
  val infer : ('a, 'b) model → 'a → 'b Distribution.t
end = struct ... end
```

## Inference algorithm

- Run a set of  $n$  independent executions
- **sample**: draw a sample from a distribution
- **factor**: associate a score to the current execution
- Gather output values and score to approximate the posterior distribution

# Importance sampling



# Importance sampling

*infer.ml*

```
module Importance_sampling = struct
  type 'a prob = { score : float; k : 'a next; value : 'a option }
  and 'a next = 'a prob → 'a prob
  and ('a, 'b) model = 'a → ('b → 'b next) → 'b next

  let sample d k prob = assert false
  let factor s k prob = assert false

  let infer ?(n = 1000) m data = assert false
end
```

Try it in BYO-PPL!

# Importance sampling

*infer.ml*

```
module Importance_sampling = struct
  type 'a prob = { score : float; k : 'a next; value : 'a option }
  ...

  let sample d k prob =
    let v = Distribution.draw d in
    k v prob
    (* draw a value *)
    (* continue *)

  let factor s k prob =
    k () { prob with score = prob.score +. s }
    (* continue with updated score *)
```



# Importance sampling

*infer.ml*

```
module Importance_sampling = struct
  type 'a prob = { score : float; k : 'a next; value : 'a option }
  ...

  let exit v prob = { prob with value = Some v }           (* return the value *)

  let infer ?(n = 1000) model data =
    let rec gen n support =
      if n = 0 then support
      else
        let k = (model data) exit in                       (* run from the start *)
        let prob = k { score = 0.; value = None; k } in    (* Run the model *)
        let value = Option.get prob.value in              (* get the value *)
        gen (n - 1) ((value, prob.score) :: support)      (* add to support *)
    in
    let support = gen n [] in
    Distribution.categorical ~support
end
```

# Coin

*coin.ml*

```
open Infer.Importance_sampling
```

```
let coin x =  
  let* z = sample (uniform ~a:0. ~b:1.) in  
  let* () = Cps_list.iter (observe (bernoulli ~p:z)) x in  
  return z
```

```
let _ =  
  let dist = infer coin [1; 1; 0; 0; 0; 0; 0; 0; 0; 0] in  
  let m, s = Distribution.stats dist in  
  Format.printf "Coin bias, mean:%f, std:%f@." m s
```

---

```
> dune exec ./examples/coin.exe
```

```
Coin bias, mean:0.247876, std:0.118921  
Beta(2+1, 8+1), mean:0.250000, std:0.120096
```

Particle filter (CPS)

---

Probabilistic Programming Languages

# Particle filter

*infer.ml*

```
module Particle_filter = struct
  include Importance_sampling

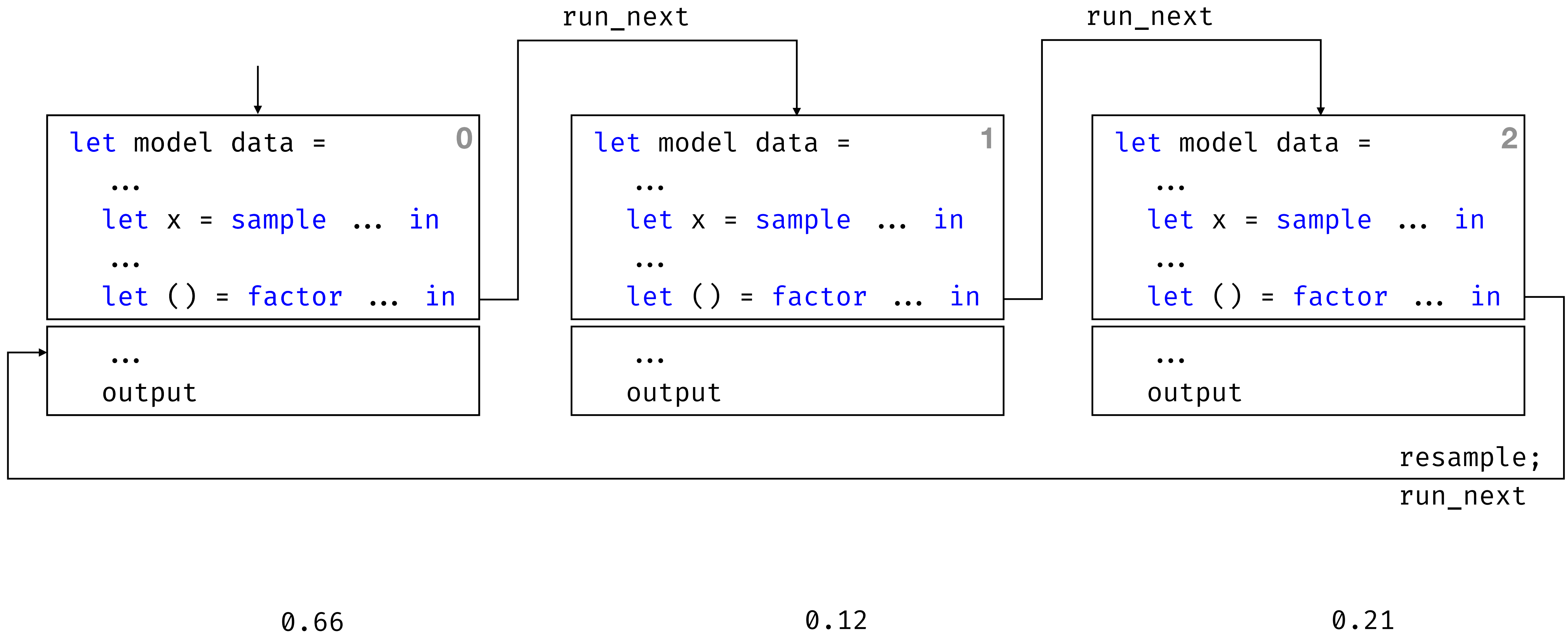
  let resample particles = assert false
  let factor s k prob = assert false

  let infer ?(n = 1000) model data = assert false
end
```

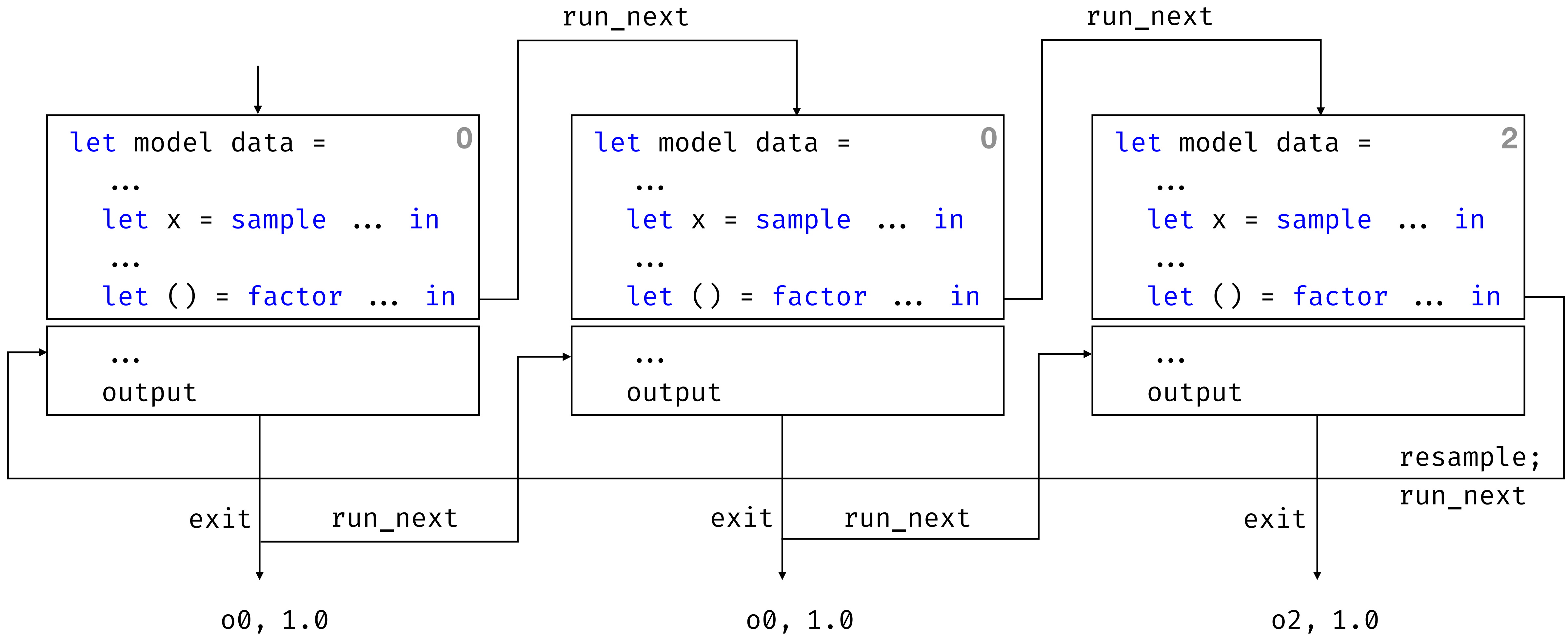
Inference algorithm : importance sampling, but...

- Add a resampling step at each **factor**
- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution

# Particle filter



# Particle filter



# Particle filter

*infer.ml*

```
module Particle_filter = struct
  include Importance_sampling

  let resample particles =
    let support = List.map (fun p → (p, p.score)) particles in      (* extract support *)
    let dist = Distribution.categorical ~support in
    let particles = List.map (fun _ → Distribution.draw dist) particles in  (* draw *)
    List.map (fun p → {p with score = 0.})                          (* reset the scores *)

  let factor s k prob =
    { prob with score = prob.score +. s; k = k () }                (* break *)
```

# Particle filter

*infer.ml*

```
module Particle_filter = struct
  ...
  let infer ?(n = 1000) model data =
    let rec gen particles =
      if List.exists (fun p → Option.is_none p.value) particles then      (* not finished *)
        let particles = List.map (fun p → p.k p) particles in             (* run next step *)
        let particles = resample particles in                               (* resample *)
        gen particles
      else particles                                                         (* finished *)
    in

    let p_init = { value = None; score = 0.; k = (model data) exit }
    let particles = List.init n (fun _ → p_init) in                        (* initial particles *)
    let support =
      let particles = gen particles in                                       (* run filter *)
      List.map (fun p → (Option.get p.value, p.score)) particles           (* extract results *)
    in
    Distribution.categorical ~support
end
```



# HMM: Hidden Markov Model (CPS)

*hmm.ml*

```
open Infer.Particle_filter
```

```
let hmm prob data = assert false
```

Try it in BYO-PPL!

# HMM: Hidden Markov Model (CPS)

*hmm.ml*

```
open Infer.Particle_filter

let hmm data =
  let rec gen states data =
    match (states, data) with
    | [], y :: data → gen [ y ] data
    | states, [] → return states
    | pre_x :: _, y :: data →
      let* x = sample prob (gaussian ~mu:pre_x ~sigma:1.0) in
      let* () = observe prob (gaussian ~mu:x ~sigma:1.0) y in
      gen (x :: states) data
  in
  gen [] data

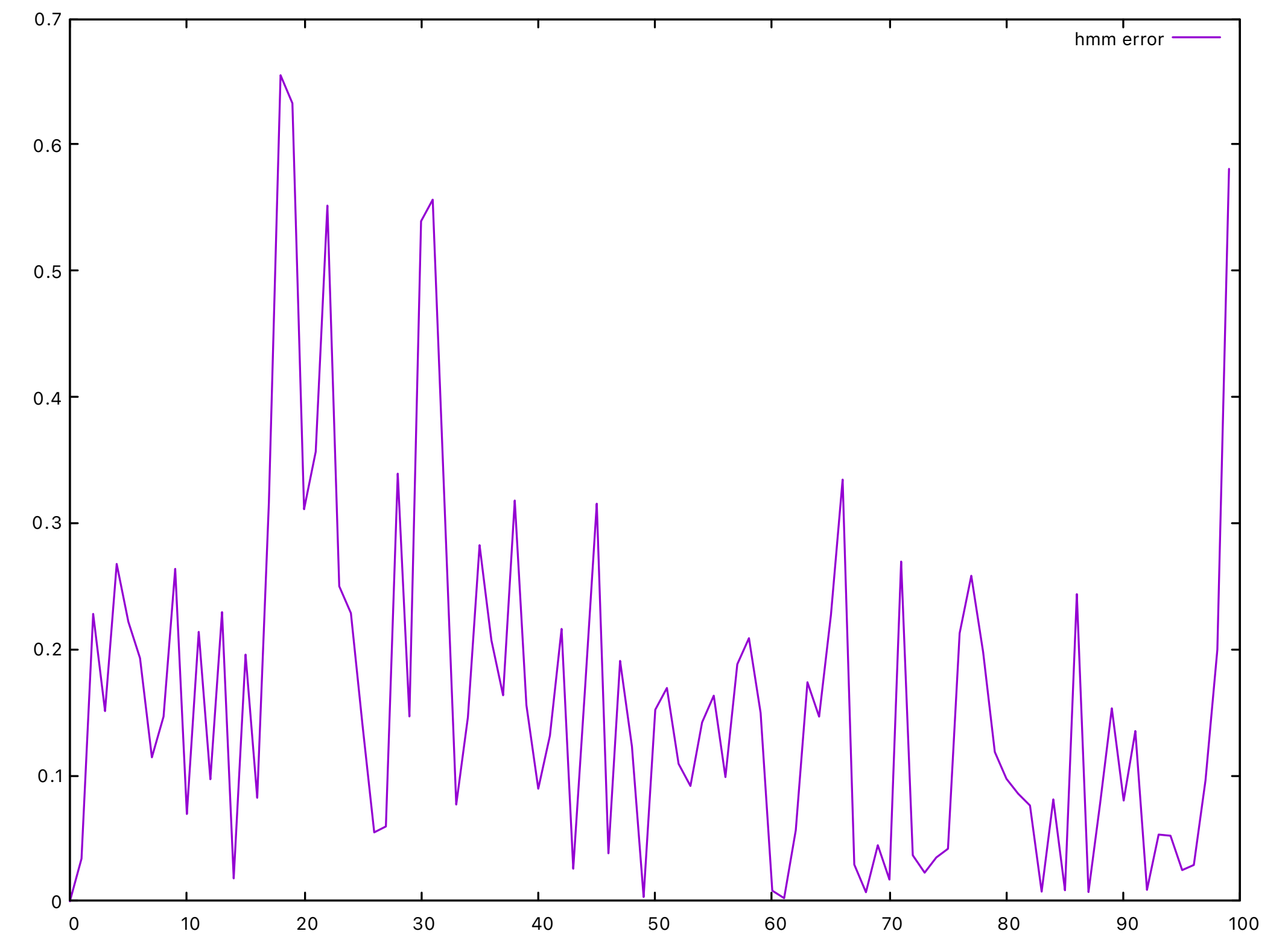
let _ =
  let data = List.init 20 (fun i → Float.of_int i) in
  let dist = Distribution.split_list (infer ~n:100 hmm data) in
  let m_x = List.map Distribution.mean dist in
  List.iter2 (Format.printf "%f >> %f") data m_x
```

# HMM: Hidden Markov Model

```
› dune exec ./hmm.exe
```

```
0.000000 >> 0.000000
1.052632 >> 0.997546
2.105263 >> 2.300316
3.157895 >> 3.289649
4.210526 >> 4.857555
5.263158 >> 4.907179
6.315789 >> 6.254198
7.368421 >> 7.208341
8.421053 >> 8.432642
9.473684 >> 8.938143
10.526316 >> 9.555007
11.578947 >> 11.098199
12.631579 >> 12.823460
13.684211 >> 13.701444
14.736842 >> 14.934314
15.789474 >> 16.115058
```

...



Exact inference: Enumeration (CPS)

---

Probabilistic Programming Languages

# Enumeration

*infer.ml*

```
module : sig
  type 'a prob
  and 'a next = 'a prob → 'a prob
  and ('a, 'b) model = 'a → ('b → 'b next) → 'b next

  val sample : 'a Distribution.t → ('a → 'b next) → 'b next
  val factor : float → (unit → 'b next) → 'b next
  val infer : ('a, 'b) model → 'a → 'b Distribution.t
end = struct ... end
```

Inference algorithm: depth first exploration

- Maintain a continuation queue
- **sample**: pick one value (and score), push the other in the queue
- **factor**: update the score
- At the end of the model, store the pair (value, score), restart from the top of the queue

Try it in BYO-PPL!

# Inference comparison

---

Probabilistic Programming Languages

# Coin

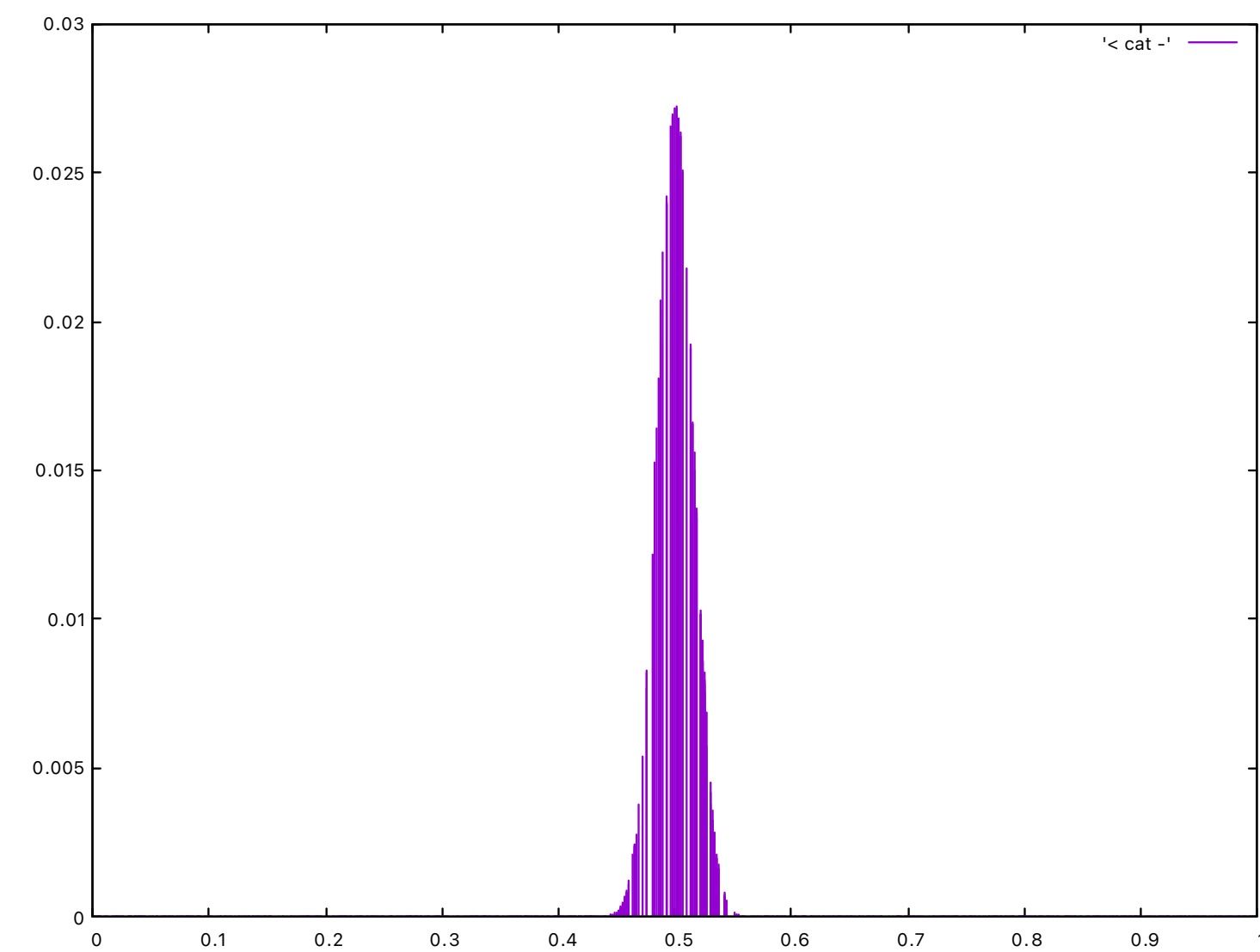
*coin.ml*

```
let _ =  
  let data = List.init 1000 (fun i → i mod 2) in  
  let dist = infer coin data in  
  plot dist
```

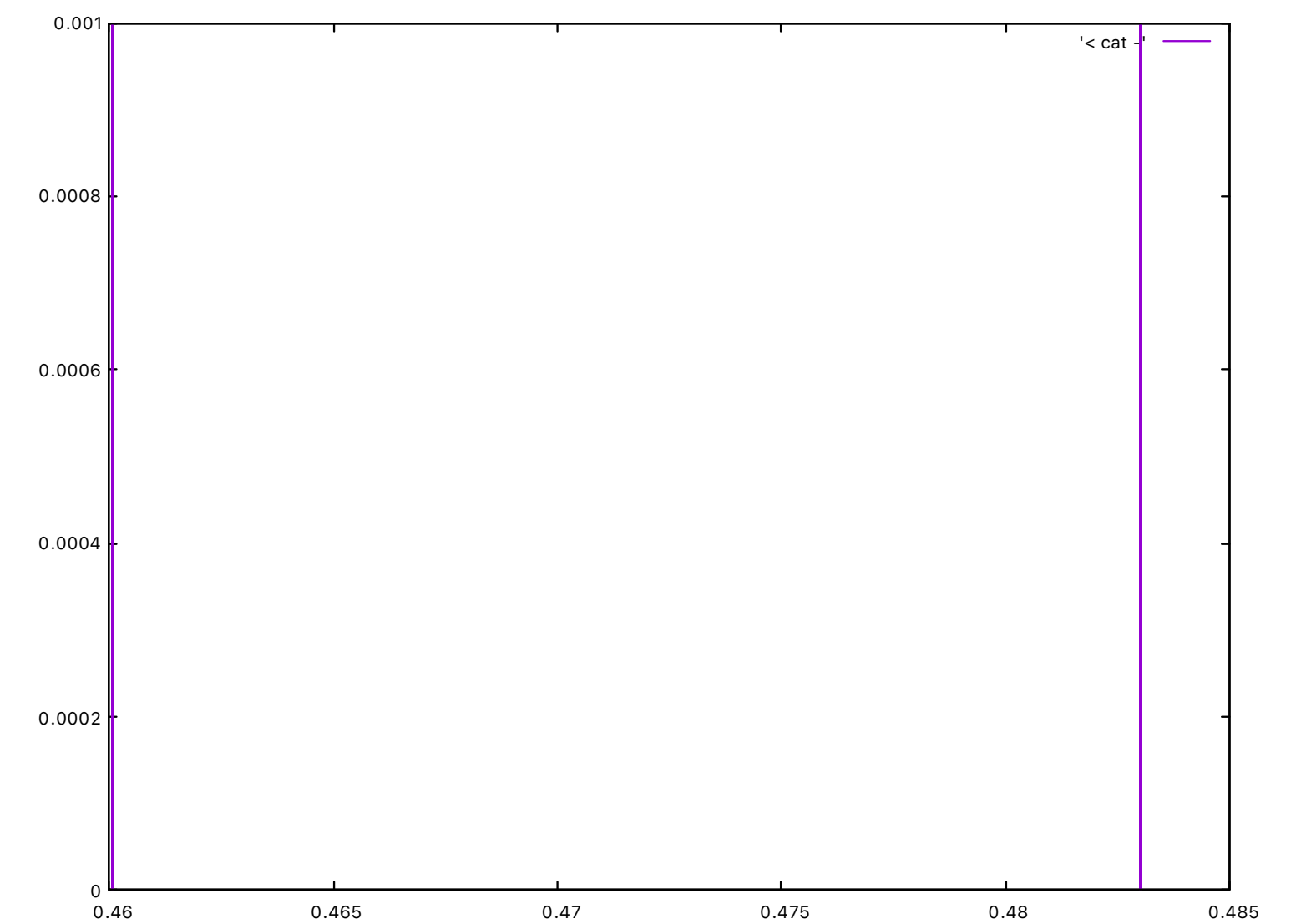
Rejection Sampling

Very (very) slow!

Importance Sampling



Particle Filter



Particle impoverishment



# HMM

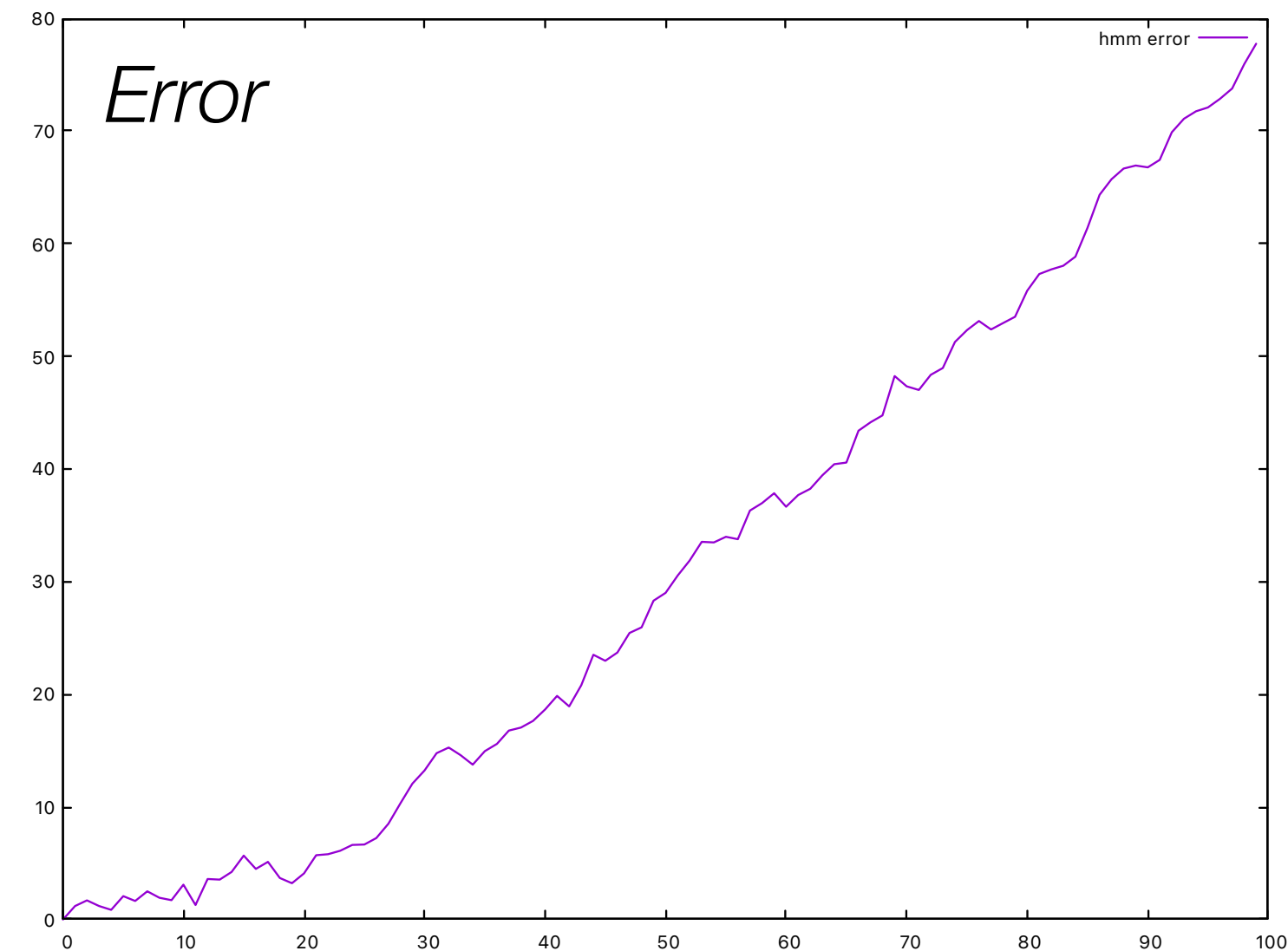
*hmm.ml*

```
let _ =  
  let data = List.init 100 (fun i → Float.of_int i) in  
  let dist = Distribution.split_list (infer hmm data) in  
  plot (error (List.rev dist) data)
```

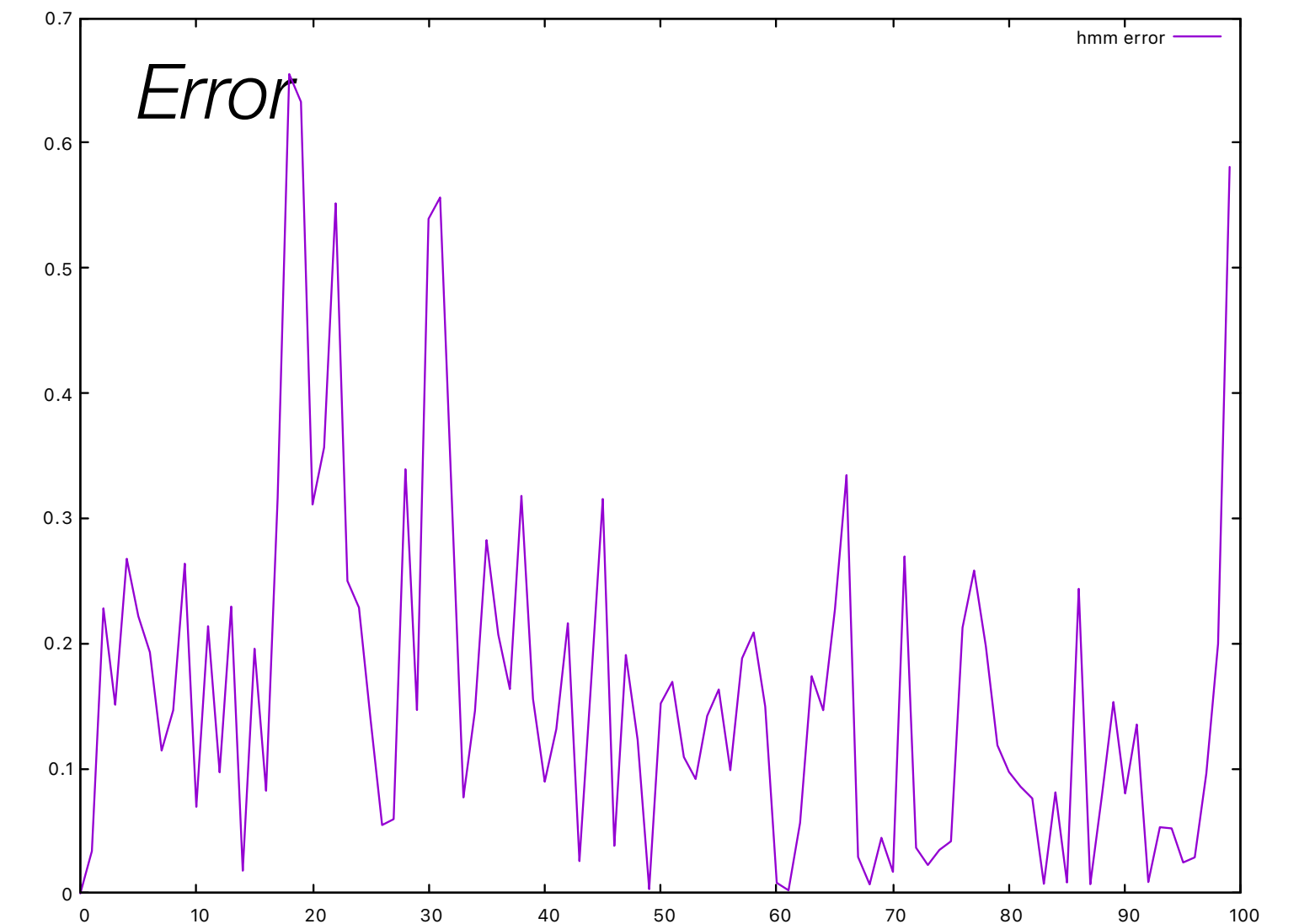
Rejection Sampling

Never terminate!

Importance Sampling



Particle Filter



# Inference limitations

## Rejection Sampling : Termination

- Only works if valid samples can be generated from the priors
- In practice: simple models with few observations
- E.g., simple discrete models.

## Importance Sampling : Weight collapse

- Score strictly decrease at each observation: eventually collapse
- In practice: continuous model to estimate fixed parameters from observations
- E.g., coin, linear regression

## Particle Filter: Particle impoverishment

- Duplicate particle without resampling fixed parameter
- In practice: high-dimensional continuous models without fixed parameters
- E.g., hmm, tracker

# Inference formalization

---

BYO-PPL

# Sampler

Probabilistic semantics  $G \vdash^P e : t$

- Expressions are interpreted as weighted samplers
- Draw random samples and track the execution weight
- Given an environment  $\gamma$ ,  $\llbracket e \rrbracket_\gamma = v, w$
- $\llbracket e \rrbracket : \Gamma \rightarrow V \times [0, \infty)$

$$\llbracket c \rrbracket_\gamma = c, 1$$

$$\llbracket x \rrbracket_\gamma = \gamma(x), 1$$

$$\llbracket \text{sample}(e) \rrbracket_\gamma = \text{draw}(\llbracket e \rrbracket_\gamma), 1$$

$$\llbracket \text{factor}(e) \rrbracket_\gamma = ( ), \llbracket e \rrbracket_\gamma$$

$$\begin{aligned} \llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket &= \text{let } v_1, w_1 = \llbracket e_1 \rrbracket_\gamma \text{ in} \\ &\quad \text{let } v_2, w_2 = \llbracket e_2 \rrbracket_{\gamma + [x \leftarrow v_1]} \text{ in} \\ &\quad v_2, w_1 \times w_2 \end{aligned}$$

Combine weights

# Importance sampling

## Inference algorithm

- Run a set of  $N$  independent executions
- `sample`: draw a sample from a distribution
- `factor`: associate a score to the current execution
- Gather output values and score to approximate the posterior distribution

$$\begin{aligned} \llbracket \text{infer}(e) \rrbracket_{\gamma} &= \text{let } [v_i, w_i = \{e\}_{\gamma}]_{1 \leq i \leq N} \text{ in} \\ &\quad \text{let } W = \sum_{i=1}^N w_i \text{ in} \\ &\quad \lambda U. \sum_{i=1}^N \frac{w_i}{W} \times \delta_{v_i}(U) \end{aligned}$$

# Particle filter

Inference algorithm : importance sampling, but...

- Add a resampling step at each **factor**
- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution

Resampling

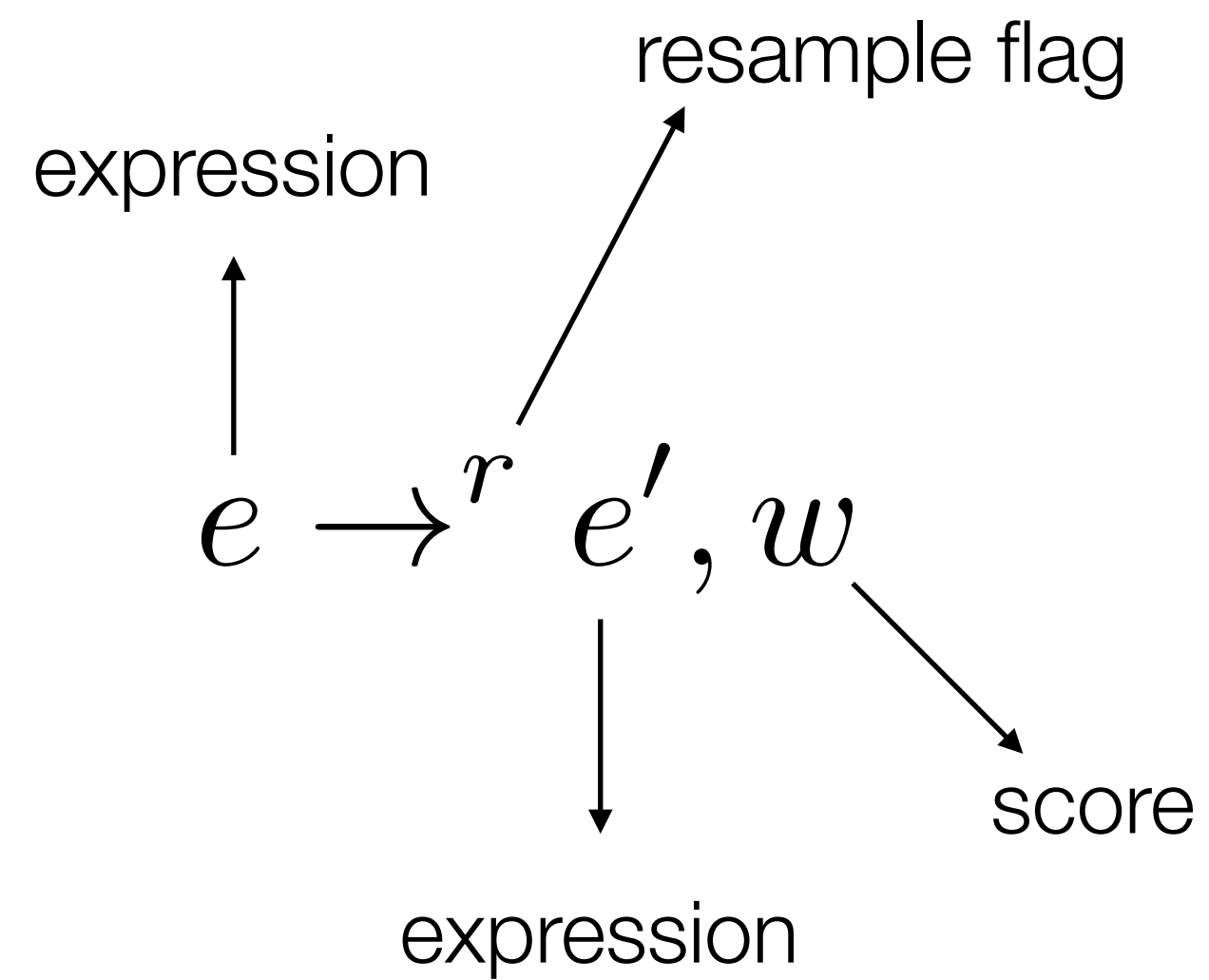
$$\begin{aligned}\llbracket e \rrbracket &\propto \llbracket \text{let } d = \text{infer}(e) \text{ in sample}(d) \rrbracket \\ \llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket &\propto \llbracket \text{let } d = \text{infer}(e_1) \text{ in let } x = \text{sample}(d) \text{ in } e_2 \rrbracket \\ \llbracket \text{infer}(\text{let } x = e_1 \text{ in } e_2) \rrbracket &= \llbracket \text{infer}(\text{let } d = \text{infer}(e_1) \text{ in let } x = \text{sample}(d) \text{ in } e_2) \rrbracket\end{aligned}$$

# Checkpoints

---

## Probabilistic Programming Languages

# Big-step semantics with checkpoints



Reduction rules: stop evaluation when  $r$  is true



# Big-step semantics with checkpoints

$$\frac{e \rightarrow^{false} true, 1 \quad e_1 \rightarrow^r e'_1, w_1}{\text{if } e \text{ then } e_1 \text{ else } e_2 \rightarrow^r e'_1, w_1}$$

$$\frac{e \rightarrow^{false} false, 1 \quad e_2 \rightarrow^r e'_2, w_2}{\text{if } e \text{ then } e_1 \text{ else } e_2 \rightarrow^r e'_2, w_2}$$

$$\frac{e_1 \rightarrow^{true} e'_1, w_1}{\text{let } x = e_1 \text{ in } e_2 \rightarrow^{true} \text{let } x = e'_1 \text{ in } e_2, w_1}$$

$$\frac{e_1 \rightarrow^{false} v_1, w_1 \quad e_2 \rightarrow^r e'_2, w_2}{\text{let } x = e_1 \text{ in } e_2 \rightarrow^r e'_2, w_1 \times w_2}$$

Combine weights

$$\frac{e \rightarrow^{false} \mu, 1}{\text{sample}(e) \rightarrow^{true} \text{draw}(\mu), 1}$$

$$\frac{e \rightarrow^{false} s, 1}{\text{factor}(e) \rightarrow^{true} (), s}$$

Stop!

# Particle filter

$$\frac{[e_i \rightarrow^{r_i} e'_i, w_i]_{1 \leq i \leq N} \quad \bigvee_{1 \leq i \leq N} r_i \quad \rho = \text{Cat} \left( \{e'_i, w_i\}_{1 \leq i \leq N} \right) \quad [\text{draw}(\rho)]_{1 \leq i \leq N} \Rightarrow \mu}{[e_i]_{1 \leq i \leq N} \Rightarrow \mu}$$

Resampling

$$\frac{[e_i \rightarrow^{false} v_i, w_i]_{1 \leq i \leq N} \quad W = \sum_{i=1}^N w_i}{[e_i]_{1 \leq i \leq N} \Rightarrow \lambda U. \sum_{i=1}^N \frac{w_i}{W} \delta_{v_i}}$$

Gather the results

$$\frac{[e]_{1 \leq i \leq N} \Rightarrow \mu}{e \Rightarrow^N \mu}$$

Launch N particles

# References

## WebPPL

Noah Goodman and Andreas Stuhlmüller

<http://webppl.org/>

## The Design and Implementation of Probabilistic Programming Languages

Noah Goodman and Andreas Stuhlmüller

<http://dippl.org/>

## An Introduction to Probabilistic Programming

Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, Frank Wood

<https://arxiv.org/abs/1809.10756>

## Embedded probabilistic domain-specific language HANSEI

Oleg Kiselyov, Chung-chieh Shan

<https://okmij.org/ftp/kakuritu/Hansei.html>